Report : Rudy, an Erlang HTTP Sever

Avneesh Vyas

September 7, 2016

1 Introduction

The report describes the various steps and procedures followed during the implementation of a rudimentary HTTP web server in Erlang using its gen-tcp socket library API. The report also provides the results of performance evaluation of the server which was done through a test client also developed in Erlang. At the end, it suggests improvements that may improve the performance and overall capability of server.

2 Main problems and solutions

The HTTP server in question being a rudimentary server should at the minimum accept HTTP GET requests from a client process running on same (localhost) or different machine (remote host) and respond with a valid HTTP response message. To fulfill this requirement, a typical server does the following:

1. Sets up a tcp listening socket on local host to listen to TCP connection requests
2. Forks a server process as soon as client attempts to connect. This is to handle messages to/from individual client. This way, server can continue listening for more client connections while other clients get handled by dedicated server processes.
3. The handler process parses the HTTP request message to read out the HTTP method, URI, headers etc.
4. Depending upon the request, the handler process responds appropriately with 'HTTP OK' or HTTP ERROR'.

Erlang's gen-tcp library provides simple functions to set up listening ports, accept connection requests and send/receive tcp messages. Also, Erlang is well adept at concurrency and it is extremely easy to spawn parallel processes to handle parallel client connections.
To improve upon the overhead of spawning process every time a client sends a tcp connect, the server code was enhanced to spawn a fixed number of multiple worker processes during initialization right after

setting up the listening port. Each worker invokes `gen_tcp:accept(Listen)` which blocks until a client attempts to connect through TCP SYN (request to connect). This is possible because Erlang 5.5.3 and higher version allows multiple parallel accept calls on the same listening socket.

The other improvement was to mimic a real world scenario of returning a physical file content in HTTP response. This was done by parsing HTTP request and extracting the URI pointing to the file. And then using Erlang's file api, the file content was read and returned in HTTP response.

```
reply({{get, URI, _}, _, _}) ->
      [$/|FileName] = URI,
      case file:open(FileName, read) of
            {ok, Binary} ->
                  Data = readData(Binary),

                  file:close(Binary),
                  http:ok([Data]);
            {error, Reason} ->
                  http:ok(io:format("Error reading file: ~s~n",
[Reason]))
      end.
```

3 Evaluation

The server was tested with a test client which generated 100 sequential HTTP GET requests and measured the total time to get response. Also, results were collected for a real world use case - the server was enhanced to return the content of a physical file in response to each GET request.

Test Scenario	Average Time (μsec)	max requests/second
Trivial reply	176200	567
Reply with 40 msec delay	4725100	21.1
Reply with file read	230800	433

Table 1: HTTP Response time for various response types

The server was also tested with a test client which generated N sequential requests on C parallel threads. The test was performed with 'smp=enabled' for Erlang on a 5-core machine. Here are the results:

	N = no. of sequential requests on one thread, C = no. Of parallel requesting threads		
No. Of server threads	N=10, C=10	N=10,C=20	N=10,C=30
1	640ms	1544ms	2138ms
4	591ms	1133ms	1674ms
10	125ms	624ms	1217ms

Table 2: HTTP Response time for parallel requests towards server with 1 or many parallel handler processes

As can be seen from the table, as the number of handler processes at the server side were increased, better throughput for incoming requests was observed.

4 Conclusions

With this assignment, I was introduced to Erlang's functional and concurrent programming paradigm. Also I learnt using Erlang's socket programming and file io library. In addition, the assignment helped me appreciate Erlang's optimization techniques related to recursion i.e tail recursion. This is not so apparent in traditional programming languages such as C/C++/Java.